J. Symbolic Computation (1998) **26**, 229–244
Article No. sy980208

A Parallel Implementation of Buchberger's Algorithm over $\mathbf{Z}_p$ for $p \leq 31991$

ALYSON A. REEVES†
Center for Computing Sciences, Institute for Defense Analyses, 17100 Science Dr., Bowie, MD 20715, U.S.A.

Gröbner bases of ideals of polynomials are known to have many applications. They have been applied to problems in commutative algebra, statistics, graph theory, robotics and differential equations. Their use as a research tool, however, is limited by their computational complexity. These two facts have inspired numerous attempts to parallelize Buchberger's algorithm to compute them.

In this paper, we describe a parallel implementation developed on the Cray T3D using the extensions to C provided by ac. The program is based on the publicly available package Macaulay which computes Gröbner bases of homogeneous ideals over $\mathbf{Z}_p$ for primes $p \leq 31991$. The efficiency is nearly 100% on up to 16 processors for moderately sized problems. Above 16 processors, the efficiency drops.

1. Introduction

Gröbner bases are known to have many applications and also to be fairly difficult to compute in many cases. These two facts have inspired numerous attempts with varying degrees of success to parallelize the algorithm to compute them (Ponder, 1988; Senechaud, 1989; Siegl, 1990;Vidal, 1990; Chakrabarti and Yelick, 1993).

The program discussed in this paper, henceforth referred to as PMac, is based on the sequential program Macaulay (Bayer and Stillman, 1994). One standard criticism of Macaulay is that it only computes Gröbner bases over the integers mod p for primes $p \leq 31\,991$. However, many problems in algebraic geometry fit into the class that Macaulay (and hence PMac) is able to solve (witness the widespread use of Macaulay).

A clear advantage to limiting the parallel algorithm to computations over $\mathbf{Z}_p$ is that the effect that a random order of computation can have on coefficient growth is eliminated. However, to use PMac for computations over $\mathbf{Z}$, one would need to employ one of the methods of lifting Gröbner bases over $\mathbf{Z}_p$ to a Gröbner basis over $\mathbf{Z}$ described in various papers (Traverso, 1988; Winkler, 1988; Gräbe, 1993; Pauer, 1992). See Section 6 for a discussion of how the method of computing over multiple primes fits in with the algorithm under discussion in this paper.

The main reasons for choosing Macaulay as a starting point were (1) familiarity with

† E-mail: aareeves@ccs.ida.nsa

0747–7171/98/080229 + 16 $30.00/0

the program, (2) availability of the source code, (3) ability to exploit the greater degree of parallelism inherent in the homogeneous algorithm than in the inhomogeneous algorithm (see Attardi and Traverso (1994)), and (4) amenability of the Macaulay memory allocation system to parallel computation.

The programming style of PMac is closest to the coarse-grained parallelism employed by Vidal (1990) and Chakrabarti and Yelick (1993) in the sense that very little synchronization occurs within the computation. There are several major differences, though. In terms of program usability, the most significant differences are that (1) Vidal's program does not delete redundant basis elements during the computation, whereas PMac does and (2) Chakrabarti and Yelick's program replicates the basis—and various versions of the basis—on all processors, whereas PMac has only one copy of a basis polynomial, and that copy occurs on only one processor. As a result, PMac scales well with respect to memory as well as with respect to speed. Part of the algorithm follows the outline for a homogeneous algorithm given in Attardi and Traverso (1994) in the sense that reduction is divided into two stages. The outline for a homogeneous algorithm given in Attardi and Traverso (1996) is even closer to the algorithm in PMac, but PMac requires more synchronization than they describe and uses the same set of processors for all phases of the algorithm, whereas their algorithm seems to require different sets of processors for each phase.

PMac was implemented on a Cray T3D having 8 Mb of memory per node. The compiler used was ac, an extension of gcc written by Carlson and Draper (1995). ac superimposes a shared memory model over the distributed memory of the machine. Its optimization abilities include enhanced instruction scheduling and prefetching.

PMac scales well. The efficiency achieved for up to 16 processors is nearly 100% for several moderately sized problems. Above 16 processors, the efficiency starts to drop to 75% or worse due to load-balancing problems, which are discussed below. In terms of true benefit over running Macaulay on an equivalent, non-parallel machine, three problems took less than 1/20 the time when run on 32 processors of the Cray-T3D. For a brief but thorough survey of related works, the reader is referred to Sawada *et al.* (1994).

In Section 2, we give definitions and an outline of the homogeneous algorithm. Section 3 describes implementation issues amd Section 4 details the parallel implementation. In Section 5 some test problems and timings are presented. Section 6 sketches a modification to the algorithm which allows us to save some time when computing the same basis over $\mathbf{Z}_p$ for multiple values of p. Finally, Section 7 gives some conclusions. The appendix lists the polynomials used in the problems of Section 5.

2. Definitions and the Homogeneous Algorithm

DEFINITION 2.1. *A polynomial is said to be* homogeneous *if all of its terms have the same degree.*

NOTE. If $F = \{f_1, \ldots, f_m\} \in k[x_1, \ldots, x_n]$ is a set of inhomogeneous polynomials, then $F^* = \{z^{d_1} f_1(x_1/z, \ldots, x_n/z), \ldots, z^{d_m} f_m(x_1/z, \ldots, x_n/z)\} \in k[x_1, \ldots, x_n, z]$, where $d_i = deg(f_i)$, is a homogeneous set called the *homogenization* of F. Setting $z = 1$ in the polynomials forming the Gröbner basis of F^* gives a Gröbner basis for F (see, for example, Becker and Weispfenning (1993, Section 10.3)).

Throughout this section, f and g will denote homogeneous polynomials in $R = k[x_0, \ldots, x_n]$, where $k = \mathbf{Z}_p$ for some prime p.

DEFINITION 2.2. *The* ideal *I generated by homogeneous polynomials $\{f_1, \ldots, f_r\}$ consists of all polynomials of the form $\sum_{i=1}^{r} h_i f_i$ where $h_i \in R$.*

DEFINITION 2.3. *Let $x^A = x_0^{a_0} x_1^{a_1} \ldots x_n^{a_n}$ denote a monomial in R. An* admissible order *$>$ on the monomials of R is a total order on the monomials such that $x^A > 1$ for all monomials x^A, and $x^A > x^B$ implies $x^A x^C > x^B x^C$ for all monomials x^A, x^B, and x^C. Given an order $>$, we denote by $in_>(f)$ (or $in(f)$ when there is no chance of confusion) the* initial (*or* leading) monomial *of f with respect to $>$, and by $in_>(I)$, the ideal generated by the initial terms of all polynomials in I. We call $f - in(f)$ the* tail *of f.*

DEFINITION 2.4. *The* s-polynomial *of the polynomials f and g is the polynomial*

$$\frac{x^A}{in(f)} f - \frac{x^A}{in(g)} g,$$

where x^A is the least common multiple of $in_>(f)$ and $in_>(g)$. (f, g) is the associated s-pair. *If $in_>(g)$ divides some non-zero term Cx^A of f (where C is the coefficient of x^A in f), the reduction of f by g is defined to be $f - Cx^A/(in(g))g$, assuming g is monic.*

DEFINITION 2.5. *A Gröbner basis with respect to the admissible order $>$ for a finite set of polynomials $\{f_1, \ldots, f_r\}$ generating the ideal I is a set of polynomials $G = \{g_1, \ldots, g_s\}$ such that g_i is in I for $i = 0, \ldots, s$ and $in_>(I)$ equals the ideal generated by $\{in_>(g_i)$ s.t. $i = 1, \ldots, s\}$.*

There are several equivalent definitions of a Gröbner basis. One definition is: G is a Gröbner basis if every s-pair of polynomials in G gives an s-polynomial which reduces to 0 mod G. This definition leads readily to an algorithm to compute Gröbner bases. This algorithm, first described by Buchberger (1965) in his thesis, proceeds as follows:

$I = \{f_1, \ldots, f_r\}$ denotes the set of input polynomials.
B denotes the list of basis elements (initially empty).
S denotes the list of s-pairs to be done.

Initialize S to the set of all s-pairs of input polynomials
While $S \neq \emptyset$ do
    Select and remove an s-pair and form the corresponding s-polynomial p
    Reduce p by the elements in B
    If the result, p', is not 0
        Reduce all elements in B by p'
        Form all s-pairs of the form (p', q) $(q \in B)$ and insert them into S
        Insert p' into B.

Restricting the input to homogeneous polynomials leads to some nice refinements of this algorithm. In particular, a partial ordering of the s-pairs becomes apparent, namely, the one induced by the degree of the corresponding s-polynomial. In addition, autoreduction (reducing the forming basis by the element to be inserted) is vastly simplified since it can only be applied to elements of the same degree as the element to be inserted,

and it can only involve the tail terms of these elements. As the homogeneous algorithm is somewhat less well known than the original algorithm, it is included below.

B, I, and S are as above.
d is initially the lowest degree of an input polynomial.

While (($S \neq \emptyset$) or ($I \neq \emptyset$))
  While ($S \neq \emptyset$)
    Remove an s-pair of degree d from S, form the s-polynomial p, and reduce p by B.
    If the result p' is not 0
      Reduce B by p'
      Form all s-pairs of the form (p', q) ($q \in B$) and insert them into S
      Insert p' into B
  While ($I \neq \emptyset$)
    remove f of degree d from I
    reduce f by B
    if the result f' is not 0
    reduce B by f'
    Form all s-pairs of the form (f', q) ($q \in B$) and insert them into S
    Insert f' into B
  $d := d + 1$

3. Implementation Issues

By modifying the sequential computer program Macaulay written by Bayer and Stillman (1994), the above algorithm (modified as described below) was implemented on a Cray T3D using ac, an extension of C developed by Carlson and Draper (1995). There were four main issues guiding the implementation: limited memory per node, updating of data, load balancing, and useability.

3.1. LIMITED MEMORY

Limited memory per node forced the distribution of many of the larger structures in the program. In particular, each polynomial is kept on only one processor, and hence the forming basis is distributed over all the processors[†]. Polynomials that are not known to be local are copied to a processor one monomial at a time. Once the processor receives a monomial, it operates on that monomial before asking for the next. The system "prefetches" the next monomial while the calculations are occurring. The list of s-pairs remaining to be processed is also distributed among the processors. The only major structure which is duplicated on all processors is the "fast monomial look-up table",

[†] There is a natural point at which a polynomial could be distributed to and left on all processors if replication of the basis were desired. The point occurs during Stage 2 (see Section 4): currently each polynomial is distributed to all processors when it is used as a reducer in Stage 2, but only the owning process keeps a copy of the polynomial. The other processors kill their copies. If replication were desired, the "killing" step could be deleted.

since distributing it slows the program down too much (by a factor of 2, at least). The limited memory also makes periodic reduction of polynomials essential.

The distribution of two of the largest structures means that the program scales well with respect to memory usage per processor. What is perhaps surprising is that it also scales well with respect to time (see Section 5).

3.2. UPDATING OF DATA

In ac, the only mechanism for ensuring data accessed by a non-local processor is current is the *barrier* which synchronizes *all* processors. Thus, to make efficient use of ac and the Cray T3D, the updating of data among processors needs to occur fairly infrequently.

For polynomials, this is done by categorizing them as stable or unstable. Stable polynomials are those that cannot be reduced further, regardless of what elements get added to the basis in the future. Unstable polynomials are those that can or could possibly be reduced by some element of the basis either now or in the future. In the homogeneous version of the algorithm, the only unstable polynomials are those of the current degree. In contrast, in the inhomogeneous algorithm, every polynomial is potentially reducible (and hence unstable) until the algorithm terminates.

The reduction of s-pairs is divided into two stages: reduction by polynomials of degree strictly less than the current degree (stable polynomials) and reduction by polynomials of the current degree (unstable). During the first stage, processors retrieve polynomials by which they need to reduce, but they never communicate the results of their reductions. That is, the current basis (set of polynomials available to reduce by) does not change. Thus, no synchronization is necessary. While there is a lot of synchronization in the second stage, there is also a lot of parallelism, as will be explained in the next section.

The one piece of information that is updated for all processors during the first stage is the list of s-pairs to be processed. Only one processor at a time may access the list (which is protected by a "critical section"), obtaining and deleting from the list the next s-pair to be processed. While this is a bottleneck at the beginning of Stage one for each degree, in practice it is rarely a problem during the rest of Stage one.

Though the s-pair list is accessed sequentially, it is formed in parallel. This is accomplished by having each processor form its own s-pair list using a subset of the computed polynomials and then linking the lists together at a later time. This will be explained in detail in the next section.

Figure 1 shows the way in which the major data structures are distributed over the processors. g1m1 denotes the first monomial of the first generator, bas1m1 denotes the first monomial of the first standard basis element, pgen1 denotes the pointer to the first generator, and pbas1 denotes the pointer to the first basis element.

3.3. LOAD BALANCING

Load balancing influenced the design of the program in two ways. First was the dynamic distribution of s-pairs. Each processor obtains the next s-pair on a first come, first serve basis. Thus some processors may end up reducing twice as many s-pairs as others.

Second was the distributed calculation of the s-pairs to be processed (detailed in the next section). Both of these load-balancing issues are vital.

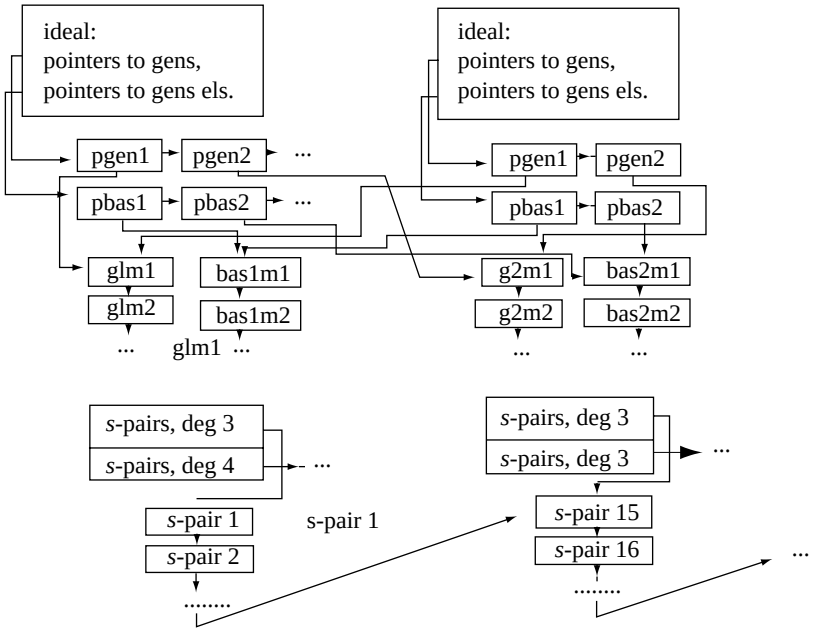

Figure 1. Distribution of data structures.

3.4. USEABILITY

The final issue is useability. In order to make the program useable to anyone familiar with Macaulay, it was kept as a "front end" to the program. Commands were added for those functions which can be computed in parallel. The commands allow the user to trace calculations, stop the computation at a particular degree, print out intermediate results, and restart the computation from that degree†. In addition, the user can compute bases over quotient rings by ideals, change the order to any allowable order in Macaulay (including product orders and weighted orders), and compute a subset of polynomials which form a Gröbner basis with respect to a subset of the variables, the rest of the variables being considered constants in the ring.

Useability was also the deciding factor in distributing rather than replicating the basis. Without the ability to compute Gröbner bases requiring more than 8 Mb to store (the Cray T3D on which this program runs has only 8 Mb per node), this program would be merely an interesting exercise.

4. Implementation Details

While the overall design of the parallel algorithm is very similar to the homogeneous algorithm outlined above, the details differ considerably.

The top level of the parallel algorithm can be described as follows:

† This capability is not included in the program timed in Section 5 due to the amount of overheads incurred while writing partial results to a file. Most often one would want this capability only for extremely long computations.

Let I, B, S and d be as previously defined.

Read in the polynomials of I and store distributively
While ($S \neq \emptyset$ or ($I \neq \emptyset$) do
  While ($S \neq \emptyset$) /* Block A */
    Form and reduce the s-polynomials of degree d corresponding to s-pairs in S
    Insert results into the basis B
    Calculate new s-pairs to be processed
  While ($I \neq \emptyset$) /* Block B */
    Reduce elements of degree d in I by the basis B and add them to B.
    Calculate new s-pairs to be processed
  $d := d + 1$

During the initialization phase, the polynomials of I are read in and stored in round-robin fashion. Thus each polynomial resides on only one processor. This part of the algorithm is entirely sequential.

The loop labelled "Block A" is the part of the algorithm that is broken into two stages, as alluded to in the previous section. During the first stage, processors obtain (distinct) s-pairs from the s-pair list S and reduce the corresponding s-polynomials as much as possible by elements in the basis B. The chain criterion (which states that the pair (f, g) can be deleted if there exists h such that $in(h) \mid lcm(in(f), in(g))$ and $lcm(in(h), in(f))! = lcm(in(h), in(g))! = lcm(in(f), in(g))$) is applied to s-pairs at this time. The basis itself is not updated during this time, so all reductions are by polynomials of degree strictly less than d, the current degree, and the hs applied in the chain criterion are also all of degree strictly less than d.

Once a processor finishes reducing its s-polynomial as much as possible, it adds the polynomial to a private "pending list" and moves on to the next s-pair of degree d on the list S. When all S-pairs of degree d have been used, the computation proceeds to Stage 2.

Stage 2 proceeds in a round-robin fashion: the processor whose turn it is "presents" a polynomial from its pending list to all the processors. All processors then reduce all basis elements (of degree d) locally residing on their processor *and all pending polynomials* by the presented polynomial and then add the initial monomial of the presented polynomial to a list of monomials to be added to their private copy of the "fast monomial look-up table". Note that by the time a polynomial is "presented", its initial monomial will never change again. In addition, it has been reduced by all of the polynomials in the current basis so far. This ensures that, at the end of Stage 2, all of the polynomials in the basis are fully autoreduced. No additional interreduction is necessary and no polynomial is ever used to reduce any other polynomial more than once.

To give some idea of the amount of parallelism being exploited, the details of /* Block A */ are given in diagramatic form in Figure 2. Rectangles with nothing in them represent processors executing (independently) the same instructions as the rectangle in that row containing instructions.

The calculation of s-pairs in Stage 3 is done in parallel, and two criteria are used to delete s-pairs *a priori*. The first is exactly the same one used in Macaulay: for $i, j < k$ if $lcm(in(f_i), in(f_k)) | lcm(in(f_j), in(f_k))$, delete (f_j, f_k). The second is the product

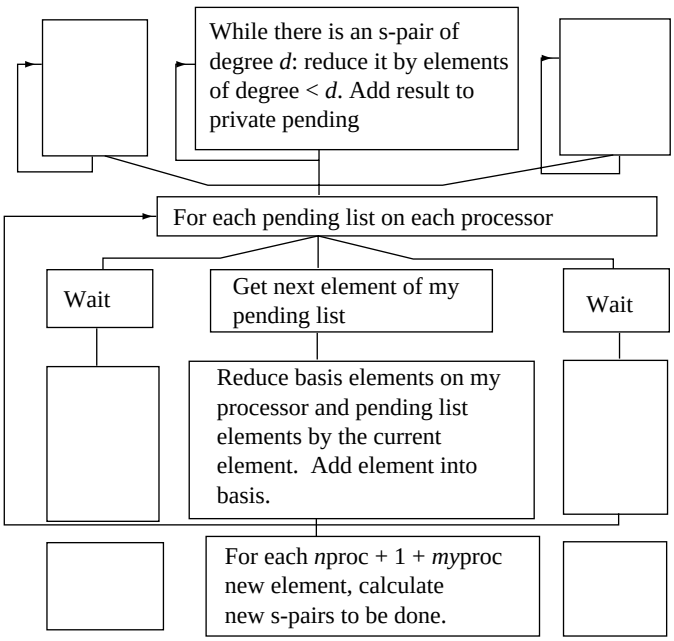

Figure 2. Details of block A.

criterion: if $lcm(in(f_i), in(f_j)) = in(f_i) \cdot in(f_j)$, delete (f_i, f_j). (The chain criterion and the product criterion were added to the sequential version of Macaulay used in the timings below.) During the computation of s-pairs processors insert, in the order computed in Stage 2, the initial terms of the new basis elements into the "fast monomial look-up table". In addition, processor k computes all s-pairs involving f_j for all $j \equiv k \ mod \ n$, where n is the number of processors. These s-pairs are inserted into a private s-pair list. When all processors have finished computing s-pairs, processor 0 links the s-pair lists together while checking for completion. Access to the list of s-pairs is through a header on processor 0 and is only allowed in "critical sections".

The *a priori* deletion of s-pairs is one instance where the randomness of the completions of calculations has an effect, since the s-pairs are calculated according to the order in which the s-polynomials were added to the basis, which, in turn, depends on which processor reduced the s-polynomial. However, differences in the number of *a priori* s-pair deletions at this stage tend to be minimized by deletions occurring during Stage 1 through the chain criterion.

5. Problems and Timings

Four questions arise when trying to measure the effectiveness of parallelizing the above procedure:

1. How does the parallel program compare with the original sequential program running on a DEC Alpha workstation?
2. How well does the program scale?
3. How much variability occurs in timings due to the inherent randomness in the program?

4. How does this program compare with the performance of other parallel programs designed to compute Gröbner bases?

In this section we present some test problems and their times. The problems are taken from various sources in the literature. Table 1 lists the times obtained for various numbers of processors and for the sequential version of Macaulay running on a DEC Alpha with the same chip running at the same speed as the Cray T3D. In the table, "high–low" is the difference between the longest and shortest times obtained in four runs of the program on the same problem. Speed-ups are computed both with respect to the time for one processor of the Cray T3D and with respect to the sequential algorithm running on the DEC Alpha. "s-pairs" gives the maximum number of s-pairs processed in any of the four runs. All problems were computed over the field GF(31991). In particular, the "cyclic-7" problem does not correspond to the problem of finding the cyclic-7 roots. Rather, the "cyclic" series of equations provided a convenient, scalable set of test problems.

The problems used are (see the Appendix for the specific polynomials):

1. Cyclic-7 equations in reverse lexicographic order (over GF(31991)): seven generators, 443 basis elements to degree 19.
2. Cyclic-7 equations in lexicographic order (over GF(31991)): seven generators, 985 basis elements to degree 211.
3. "Random 1" in elimination order (Grassman *et al.*, private communication): five generators, 1183 basis elements up to degree 44.
4. "Random 2" in elimination order (Grassman *et al.*, private communication): four generators, 969 basis elements up to degree 45.
5. 4×4 commuting matrices computed in a product order given in Hreinsdóttir (1994): 15 generators, 294 basis elements to degree 8.
6. "Rees A" in elimination order (Caboara *et al.*, 1996), 16 generators, 639 basis elements to degree 37.
7. "Aldo" in elimination order (Caboara *et al.*, 1996), 36 generators, 2045 basis elements to degree 15.
8. "Five generic equations in five variables of degree 4" in reverse lexicographic order, 206 basis elements to degree 17.

The data in the table shows that the program *scales* almost linearly from 1 to 16 processors for several moderately sized problems, though clearly input is a deciding factor in how well the program scales. "Aldo" scales linearly up to 32 processors. On 64 processors, "Aldo" took 702.2 seconds, representing a speed-up factor of 46 which, though sublinear, is still respectable. In contrast, for "Rees A" and "cyclic 7 in lexicographic order", the point at which the scaling becomes sublinear is much earlier—at four processors. The primary difficulty is load balancing. Above a certain number of processors there are not enough s-pairs per processor per degree to make the *average* time spent reducing s-pairs on each processor roughly equal. In addition, the "autoreduction" phase (the reduction of degree d polynomials by degree d polynomials) starts to take more time in proportion to the time spent on reduction of s-pairs by polynomials of lower degree. Since the autoreduction phase requires more synchronization of processors, this slows the program down.

For most problems, there is very little variation in the timings and in the number of s-pairs processed (not eliminated *a priori*), despite the inherent randomness, though

Table 1. Timings in seconds.

	DEC	1 Process	2 Processes	4 Processes	8 Processes	16 Processes	32 Processes
1.	636.0	782.56	402.81	208.23	107.63	54.60	30.96
Speed-up	1.23	1.00	1.94	3.76	7.27	14.33	25.28
	1.00	0.81	1.58	3.05	5.91	11.65	20.54
High–low		0.11	0.53	0.32	1.13	0.25	0.26
s-pairs	2192	2192	2192	2192	2192	2192	2192
2.	10740.00	14320.84	7741.01	5071.21	3717.05	2930.67	2584.14
Speed-up	1.33	1.00	1.85	2.82	3.85	4.89	5.54
	1.00	0.75	1.39	2.12	2.89	3.66	4.16
High–low		0.79	6.21	9.84	5.26	9.15	2.60
s-pairs	5505	5505	5505	5505	5505	5505	5505
3.	1879.00	1989.58	1062.42	552.22	293.95	158.44	100.50
Speed-up	1.06	1.00	1.87	3.60	6.77	12.56	19.80
	1.00	0.94	1.77	3.40	6.39	11.86	18.70
High–low		0.12	5.15	3.63	6.08	3.96	2.44
s-pairs	3379	3379	3381	3383	3383	3382	3383
4.	5862.00	5997.58	3189.89	1664.71	875.05	468.10	291.09
Speed-up	1.02	1.00	1.88	3.60	6.85	12.81	20.60
	1.00	0.98	1.84	3.52	6.70	12.52	20.14
High–low		1.57	14.67	9.86	17.82	6.02	4.36
s-pairs	5286	5286	5286	5287	5285	5285	5285
5.	246.00	415.26	202.29	101.56	52.05	28.42	17.02
Speed-up	1.69	1.00	2.05	4.09	7.98	14.61	24.40
	1.00	0.59	1.22	2.42	4.73	8.66	14.45
High–low		0.08	0.44	0.54	0.40	0.26	1.83
s-pairs	2769	2769	2770	2770	2769	2769	2768
6.	472.00	681.05	348.14	194.71	134.68	111.83	103.39
Speed-up	1.44	1.00	1.96	3.50	5.06	6.09	6.59
	1.00	0.69	1.36	2.42	3.50	4.22	4.57
High–low		1.19	3.52	9.24	7.90	0.97	2.05
s-pairs	8702	8702	8703	8703	8703	8703	8702
7.	21376.00	*	16181.67	8098.61	4040.66	2014.62	1017.11
Speed-up	1.52	*	2.00	4.00	8.02	16.09	31.86
	1.00	*	1.32	2.64	5.29	10.61	21.02
High–low		*	53.72	6.84	16.62	19.66	37.12
s-pairs	67298	*	67298	67298	67298	67298	67298
8.	47.00	65.73	34.18	17.96	9.49	5.12	3.29
Speed-up	1.40	1.00	1.92	3.66	6.93	12.83	19.95
	1.00	0.72	1.38	2.62	4.95	9.18	14.29
High–low		0.06	0.21	0.15	0.08	0.04	0.03
s-pairs	752	752	752	752	752	752	752

∗ Not enough memory.

there is some variation in how many are eliminated by the chain criterion vs. how many are eliminated by the others.

It is difficult to compare this implementation with other parallel programs designed to compute Gröbner bases due to the variations in the problems each is able to solve. The program of Vidal (1990) performs as well or better with respect to speed-up but is almost

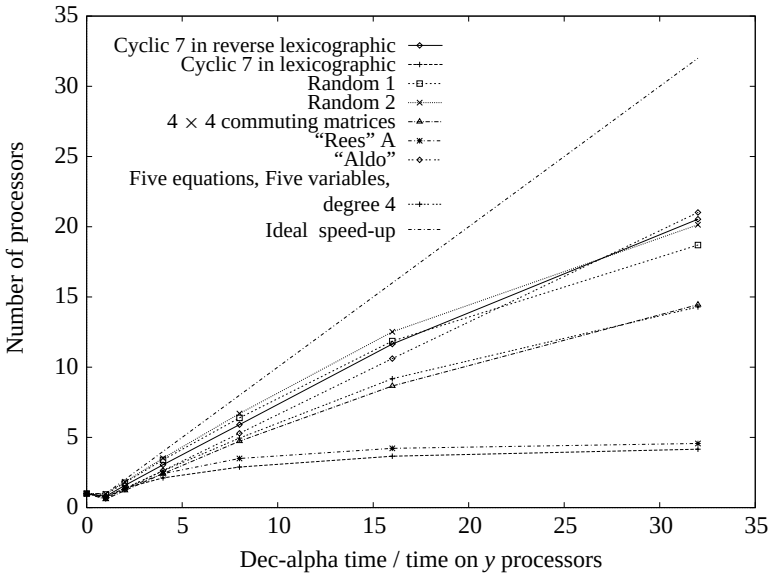

Figure 3. Speed-up with respect to sequential algorithm.

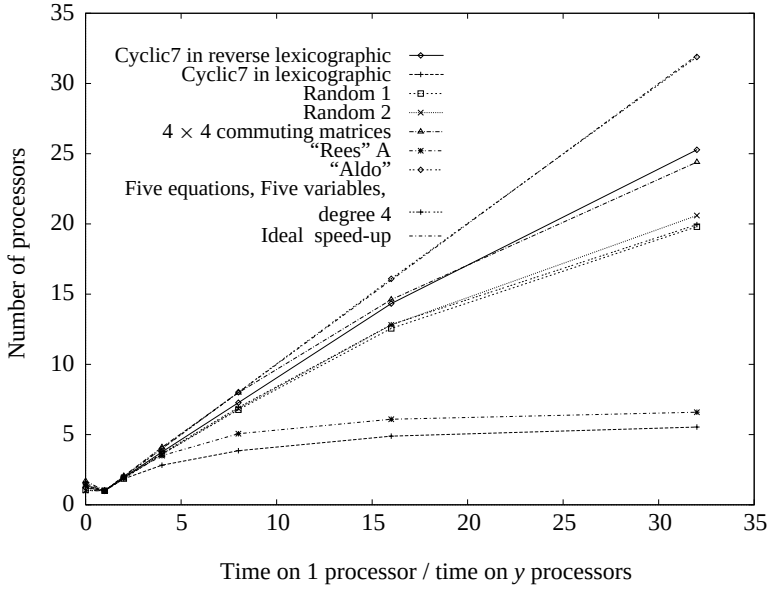

Figure 4. Speed-up with respect to parallell algorithm.

certainly more restricted in the size of the problems it can handle. In addition, the Encore machine Vidal used is a shared-memory machine. While the program of Chakrabarti and Yelick (1993) on the CM-5 runs on a distributed-memory machine, their replication of the basis limited the size of the problems they could handle. Also, they were working in characteristic 0 and it is unclear what effect this had on the timings since, on the one hand, the choice of which s-polynomial to reduce next becomes more of an issue in

characteristic 0 (due to potential coefficient growth) while on the other hand, more time spent computing coefficients means less communication time between processors. Melenk and Neun (1988) achieved a two-fold speed-up but on only two processors. Ponder (1988), Siegl (1990) and Senechaud (1989) each achieved a 1:2 ratio of speed-up to processors using various methods and with various restrictions on the problems. Sawada *et al.* (1994) achieved an eight-fold speed-up on 64 processors, but only a three-fold speed-up over a Sun Sparc Server 490. From this it maybe concluded that PMac is the most practical of the parallel implementations in terms of speed and flexibility. However, PMac has the same limitations Macaulay does: it only handles homogeneous problems (not a restriction for the most part) and the coefficients are always computed mod some large prime. As mentioned at the beginning of Section 2, it is always possible to homogenize an ideal, compute a Gröbner basis, and then dehomogenize the result to get the Gröbner basis of an inhomogeneous ideal. Thus, aside from the extra memory used by the larger, homogeneous basis, the restriction to homogeneous problems does not limit the class of problems the program can solve. The coefficient restriction is more of a problem. The next section discusses how an extension to PMac could be built to handle computations over characteristic 0 through lifting. More experiments would need to be performed to determine whether the algorithm is faster than computing directly over $\mathbf{Z}$.

6. Re-use of Information

Traverso (1988) introduced the concept of a Gröbner trace and showed that it can be used to lift results from $\mathbf{Z}_p$ to $\mathbf{Q}$. Since then several authors (Winkler, 1988; Gräbe, 1993; Pauer, 1992) have improved on his method. Gräbe's method requires the computation of Gröbner bases for the same ideal over multiple primes. Since Macaulay and hence PMac, is limited to computations over primes ≤ 31991, knowing to what extent the above algorithms can be applied using PMac may be desirable.

Like Macaulay, PMac has the ability to trace a computation. However, unlike Macaulay, there is a certain amount of randomness that occurs during a computation performed by PMac. Thus, in order to repeat a calculation exactly (or over a different prime), one needs to do some extra work. Fortunately, the randomness in PMac is essentially limited to Stage 2 where the interreduction of polynomials of the same degree takes place. This randomness can be removed by storing a list of pairs of initial terms corresponding to s-pairs which yield (non-zero) basis elements along with a number corresponding to the order in which actual initial terms of the corresponding s-polynomials are computed in Stage 2. To re-use this information, the list is read in, polynomials used to form s-polynomials are found through the "fast monomial look-up table", and the result of an s-pair computation at the end of Stage 1 is stored in an array in the spot corresponding to its number. In Stage 2, polynomials are taken in the order dictated by the array. Everything else proceeds as before. If a user only wanted to save work and did not actually care about keeping the trace exactly the same, just the list of pairs of initial terms could be stored.

While this scheme as not been implemented, a similar scheme (somewhat simpler—and just making use of the s-pair list, not the entire trace) has been implemented in the *sequential* program and the resulting time reduction tends to be a factor of between 2 and 5.

7. Conclusions

Several conclusions can be drawn from the results of this experiment. For this application, the parallel paradigm presented by ac is very effective; much more so, in fact, than the majority of paradigms used in previous parallel implementations of this algorithm. The success of this paradigm with this application is attributable to several factors. First and foremost, ac gives the programmer easy access to the hardware capabilities of the T3D that make it possible to fetch data from one processor to another with very few overheads. The program makes use of this ability by fetching polynomials one monomial at a time and then processing that monomial before requesting another. Through enhanced instruction scheduling this scheme allows the ac compiler to amortize the extra time for a distributed access by prefetching the next monomial while the current monomial is being manipulated.

A second factor in the success of program is the division of the program into large sections within which accessed data is known never to be modified. This eliminates the need for block-and-send- or block-and-receive-type protocols.

Nevertheless, the amount of parallelism exploitable in this algorithm is certainly limited (witness the less than optimal speed-up obtained in going from 16 to 32 processors in most of the problems). The main problem is load-balancing: too much time is wasted while some processors wait for others to finish an s-pair of a particular degree. In fact, for problems in which the number of s-pairs per degree is low (e.g. "Rees A" and "cyclic-7 in lexicographic order"), the algorithm behaves extremely poorly for just this reason. One possible solution to this problem would be to allow waiting processors to partially reduce s-pairs of higher degree than the current degree. However, this would seem to lead to more of an imbalance later in the program, and it would certainly increase the amount of randomness.

A second problem is the amount of synchronization necessary during the reduction of the pending lists. Vidal's solution to this problem was to leave this phase until the end. With limited memory per node, this is not a viable option for PMac. In fact, for very large problems it is sometimes necessary to reduce pending lists two or more times during the computations for a given degree.

While such problems are of theoretical interest, the practical fact remains that this program is capable of reducing the computation time for a large problem from 5–6 hours down to 15–30 minutes. Implementation of recent improvements to the Gröbner basis algorithm could further reduce this time. When the next problem to try depends on the results from the previous problem, this represents a significant increase in the number of trials that can be run and speed at which the research can proceed.

The program is available by request.

References

Attardi, G., Traverso, C. (1994). A strategy–accurate parallel Buchberger algorithm. In PASCO'94, volume 5 of Lecture Notes Series in Computing, Linz, Austria, Spt.

Attardi, G., Traverso, C. (1996). Strategy–Accurate Parallel Buchberger Algorithms. *J. Symb. Comput.*, **21**, 411-425.

Becker, T., Weispfenning, V. (1993). *Gröbner Bases: A Computational Approach to Commutative Algebra*. Springer-Verlag.

Bayer, D., Stillman, M. (1994). Macaulay: a system for computation in algebraic geometry and commutative algebra, 1982–1994. Source and object code available for Unix and Macintosh computers. Contact the authors, or download from math.harvard.edu via anonymous ftp.

Buchberger, B. (1965). An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Polynomial Ideal. PhD thesis, Universitat Innsbruck, Institut fur Maathematik. German.

Caboara, M., De Dominicis, G., Robbiano, L. (1996). Multigraded Hilbert Functions and the Buchberger Algorithm. In ISSAC '96.

Carlson, W., Draper, J. (1995). AC for the T3D. Technical Report, Center for Computing Sciences.

Chakrabarti, S., Yellick, K. (1993). Implementing an irregular application on a distributed memory multiprocessor. In *Principles and Practices of Parallel Programming*, May 1993.

Gräbe, H.-G. (1993). On lucky primes. *J. Symb. Comput.*, **15**, 199–209.

Grassmann, H., Greuel, H.-M., Martin, B., Neumann, W., Pfister, G., Pohl, W., Schönemann, H., and Siebert, T. Standard bases, syzygies and their implementation in SINGULAR. (private communication).

Hreinsdóttir, F. (1994). A case where choosing a product order makes the calculation of a Groebner basis much faster. *J. Symb. Comput.,* **18**, 373-378.

Melenk, H., Neun, W. (1988). Parallel Polynomial operations in the large buchberger algorithm. *Computer Algebra and Parallelism*, Workshop at the TIM3 Laboratory, University of Grenoble, France, Academic Press, London, June, 143-158.

Pauer, F. (1992). On lucky ideals for Gröbner basis computation. *J. Symb. Comput.*, **14**, 471–482.

Ponder, C.G. (1988). Evaluation of "Performance Enhancements" in Algebraic Manipulation Systems. PhD. Thesis, Computer Science Division, University of California, September.

Sawada, H., Terasaki, S., Aiba, A. (1994). Parallel computation of Gröbner bases on distributed memory machines. *J. Symbol. Comput.*, **18**, 207–222.

Senechaud, P. (1989). Implementation of a parallel algorithm to compute a Gröbner basis boolean polynomials. *Computer Algebra and Parallelism.* Academic Press, 159–166.

Siegl, K. (1990). Gröbner basis computation in STRAND: A case study for concurrent symbolic computation in logic programming languages. DIPLOMA Thesis, RISC-LINZ, November.

Traverso, C. (1988). Gröbner trace algorithms. *Proceedings ISSAC'88, Lecture Notes in Comp. Sci.* **358**, 125–138.

Vidal, J.-P. (1990). The computation of Gröbner bases on a shared memory multiprocessor. Proc. DISCO '90.

Winkler, F. (1988). A p-adic approach to the computation of Gröbner Bases. *J. Symb. Comput.*, **6**, 287–304.

Appendix: polynomials and orders used in the problems

"cyclic-7"

variables: a–g, z

order: either reverse lexicographic or lexicographic

ideal:

$a+b+c+d+e+f+g,$

$a*b+b*c+c*d+d*e+e*f+f*g+g*a,$

$a*b*c+b*c*d+c*d*e+d*e*f+e*f*g+f*g*a+g*a*b,$

$a*b*c*d+b*c*d*e+c*d*e*f+d*e*f*g+e*f*g*a+f*g*a*b+g*a*b*c,$

$a*b*c*d*e+b*c*d*e*f+c*d*e*f*g+d*e*f*g*a+$
$e*f*g*a*b+f*g*a*b*c+g*a*b*c*d,$

$a*b*c*d*e*f+b*c*d*e*f*g+c*d*e*f*g*a+$
$d*e*f*g*a*b+e*f*g*a*b*c+f*g*a*b*c*d+g*a*b*c*d*e,$

$a*b*c*d*e*f*g-z^7$

"Random 1"

variables: w, x, y, z

order: eliminate w

ideal:

$47x^7y^8z^3+91x^7*y^4*z^7+28x^3*y^6*z^8+63*x^2*y,$

$21x^3 * y^2 * z^{10} + 57x * y^7 * z + 15x^3 * y * z^5 + 51x * y^3 * z^3$,
$32x^7 * y^4 * z^8 + 53x^6 * y^6 * z^2 + 17x^3 * y^7 * z^2 + 74x * y^5 * z$,
$81x^{10} * y^{10} * z + 19x^3 * y^5 * z^5 + 79x^5 * z^7 + 36x * y^2 * z^3$,
$32x^{10} * y^9 * z^6 + 23x^5 * y^8 * z^8 + 21 * x^2 * y^3 * z^7 + 27y^5 * z$

"Random 2"
variables: w, x, y, z
order: eliminate w
ideal:
$57x * y^2 * x^3 + 28x * y^7 * z + 63x * y^5 * z^4 + 91x^2 * y^3 * z^7 + 47x^7 * y^8 * z^3$,
$51x^3 * y^2 * z^{10} + 21x^3 * y^6 * z^8 + 15x^7 * y^4 * z^8 + 32x^{10} * y^{10} * z + 74x^{10} * y^9 * z^6$,
$53x^2 * y + 17x * y^3 * z63 + 23x^3 * y^5 * z^5 + 21x^6 * y^6 * z^2 + 32x^5 * y^8 * z^8$,
$19y^5 * z + 36y * z^5 + 81x^3 * y^7 * z^2 + 79x^5 * z^7 + 27x^7 * y^4 * z^7$

"commuting 4×4"
variables: $x[1] - x[16]y[1] - y[16]$
order: product $x[1] - x[8], x[9] - y[4], y[5] - y[16]$

Given matrix $X(4 \times 4)$

$x[1]$	$x[9]$	$y[5]$	$y[7]$
$x[15]$	$x[3]$	$x[11]$	$x[13]$
$y[11]$	$y[1]$	$x[5]$	$y[9]$
$y[13]$	$y[3]$	$y[15]$	$x[7]$

matrix $Y(4 \times 4)$

$x[2]$	$x[10]$	$y[6]$	$y[8]$
$x[16]$	$x[4]$	$x[12]$	$x[14]$
$y[12]$	$y[2]$	$x[6]$	$y[10]$
$y[14]$	$y[4]$	$y[16]$	$x[8]$

ideal: polynomials in the matrix XY-YX
(with the last one deleted since it's extraneous)

"Rees A"
variables: $ty[1] - y[16]x[1,1] - x[4,4]$
order: eliminate t

Given matrix $M(4 \times 4)$

$x[1,1] + x[2,4] + x[2,1]$	$x[1,2] + x[2,2] + x[1,4]$	$x[1,3] + x[2,3] + x[2,4]$	$x[1,4] + x[2,4] + x[1,1]$
$x[2,1]$	$x[2,2]$	$x[2,3]$	$x[2,4]$
$x[3,1]$	$x[3,2]$	$x[3,3]$	$x[3,4]$
$x[4,1]$	$x[4,2]$	$x[4,3]$	$x[4,4]$

and the ideal $Y =< y[1], y[2], ..., y[16] >$
ideal: 3×3 minors of M each multiplied by t minus the ideal Y.

"Aldo":
variables: $x[0] - x[9]y[1] - y[36]$
order: eliminate $x[0] - x[9]$
ideal:

```
k = 1;
for (i = 1; i < 9; i++){
    for (j = i; j < 9; j++){
        x[i] * x[j] - x[i - 1] * x[j + 1] + y[k];
        k++;
    }
}
```

Originally received 28 July 1997
Accepted 17 February 1998